VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Implementace knihoven pro nástroj Kaira

Implementation of Libraries For the Tool Kaira

2015

Marek Večerka

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Marek Večerka**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Implementace knihoven pro nástroj Kaira
Implementation of Libraries For the Tool Kaira

Zásady pro vypracování:

Kaira je nástroj určený pro vytváření distribuovaných aplikací. Kaira z modelů vycházejících z barevných Petriho sítí generuje C++ MPI aplikace. Hlavním cílem práce bude implementace knihoven. Cíle práce lze shrnout v těchto bodech.

1. Seznamte se s nástrojem Kaira.
2. Sestavte množinu standardních problémů z oblasti distribuovaného programování, které by bylo vhodné umístit do vytvářených knihoven.
3. Rozšiřte nástroj Kaira o připravené knihovny.
4. Demonstrujte funkčnost řešení na netriviálním příkladě (příkladech).

Seznam doporučené odborné literatury:

[1] S. Böhm, M. Běhálek: Usage of Petri nets for high performance computing, Functional high-performance computing, ser. FHPC '12. New York, NY, USA: ACM, 2012, pp. 37–48.
[2] Domovské stránky projektu Kaira - http://verif.cs.vsb.cz/kaira/

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. Marek Běhálek, Ph.D.**

Datum zadání: 01.09.2013
Datum odevzdání: 07.05.2015

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 6. května 2015 .....

Rád bych touto cestou poděkoval Ing. Marku Běhálkovi, Ph.D. za vedení mé bakalářské práce, cenné rady a odborný dohled. Také děkuji za cenné rady celému týmu, který pracuje na vývoji nástroje Kaira. V neposlední řadě bych rád poděkoval všem, kteří mě podporovali během studia a vzniku této práce.

Abstrakt

Tato práce je zaměřena na rozšíření nástroje KAIRA (http://verif.cs.vsb.cz/kaira/), který je dostupný jako open-source pod licencí GPL(GNU General Public License). Cílem mé práce je vytvořit knihovnu, která se skládá z pojmenovaných modulů, které mají svou specifickou funkčnost a jsou zaměřeny na konkrétní problematiku v oblasti paralelního programování z pohledu vizuálního návrhu.

Hlavní přínos mé práce spočívá ve snaze minimalizovat situace, se kterými se programátor setkává během konstrukce vizuálního modelu, kdy dochází ke skutečnosti, že programátor je nucen opakovaně vytvářet již kdysi vytvořené části řešení, sekvence míst, hran a přechodů, což snižuje jeho efektivitu při řešení daného problému.

Klíčová slova: Kaira, MPI, paralelní programování, vizuální návrh, modul

Abstract

This work is focused on expansion tool KAIRA (http://verif.cs.vsb.cz/kaira/), which is available as open-source under GPL (GNU General Public License). The goal of my work is create library that contains named modules with specific functions and focus on the issues about parallel programing from the perspective of visual draft.

The main benefit of my work is effort to minimize situations with which the programmer encounters during construction a visual model, when there is the fact, the programmer is forced to re-create already had formed part of the solution, sequences places, edges and transitions, which reduces its effectiveness in solving the problem.

Keywords: Kaira, MPI, parallel programing, visual draft, module

Seznam použitých zkratek a symbolů

$	–	Dolar, označení vstupního místa modulu.
#	–	Hash, označení uživatelského tagu.
C#	–	C Sharp, vysokoúrovňový objektově orientovaný programovací jazyk vyvinutý firmou Microsoft.
C++	–	Multiparadigmatický programovací jazyk, který vyvinul Bjarne Stroustrup.
GUI	–	Graphical user interface
ID	–	IDentification, zkratka pro identifikaci.
int	–	Integer, datový typ reprezenetující celé číslo.
MPI	–	Message Passing Interface
OOP	–	Object-oriented programming
PTP	–	Project-to-program
RPC	–	Remote procedure call

Obsah

Seznam obrázků

Seznam výpisů zdrojového kódu

1 Úvod

Nástroj Kaira je vývojové prostředí zaměřené na tvorbu aplikací využívající knihovnu MPI. Vývoj aplikací se provádí především vizuální formou, která je inspirovaná Petriho sítěmi. Při návrhu nových aplikací dochází k situacím, kdy je vývojář nucen opakovaně vytvářet sekvence míst, hran a přechodů, které již v minulosti vytvořil a tyto sekvence použil u jiných navržených projektů.

Tato práce je zaměřena na rozšíření nástroje Kaira o knihovnu, která obsahuje pojmenované moduly. Moduly jsou reprezentovány sekvencemi míst, hran a přechodů, se kterými se uživatel nejčastěji setkává během vývoje nových aplikací a tyto moduly je následně možné vkládat na požadované místa uživatelské sítě, což značně urychlí samotný vývoj aplikací.

Kapitola 2 je věnována úvodu do Petriho sítí. V této kapitole jsou představeny Petriho sítě jako nástroj pro návrh a modelování dynamických systémů, dále jejich struktura, varianty a výhody jejich použití.

Knihovna MPI a základní druhy komukací, které probíhají mezi procesy v této knihovně jsou uvedeny v kapitole 3.

V kapitole 4 je představen samotný nástroj Kaira a to především jeho vlastnosti, struktura a ukázka grafického uživatelského rozhraní.

Specifikace problému, se kterým se uživatel setkává během vizuálního návrhu nových aplikací a tedy hlavní motivace, pro vznik této práce je detailněji popsána v kapitole 5.

Kapitola 6 popisuje modul a s ním spojené vlastnosti, které je možné využít během samotného návrhu modulu. Knihovna, která obsahuje navrhnuté moduly je uvedena v kapitole 7. U navržených modulů je popsána jejich funkce, struktura a také je uvedena vizuální podoba modulu.

Mechanismus, který obstarává vložení modulu na požadované místa uživatelské sítě je reprezentován ve formě průvodce. Průvodce vložení modulu do sítě a jeho jednotlivé kroky, jsou popsány v kapitole 8. A následně funkčnost navrženého řešení je demonstrována na příkladu v kapitole 9.

Na závěr mé práce jsem v kapitole 10 uvedl možné rozšíření navrhnutého řešení a v kapitole 11 je popsána myšlenka dalšího možného využití navrženého mechanismu práce s moduly během vizuálních návrhů rozsáhlých aplikací.

2 Petriho sítě

Petriho sítěmi jsou označeny matematické modely, které slouží k grafickému popisu paralelismu informační závislosti a konfliktů moderních distribuovaných systémů. Dále jsou Petriho sítě používány při návrhu aplikací, analýz a modelování paralelních a distribuovaných systémů v oblasti databázových systémů, překladačů, v telekomunikacích, nebo při popisu automatizovaných průmyslových systémů.

Petriho sítě jsou srozumitelné a také snadno analyzovatelné, tyto vlastnosti jsou dané jednoduchostí Petriho sítí. Model Petriho sítě je tvořen místy (places), která obsahují informaci o stavu ve formě žetonů (tokens) a přechody (transitions), které umožňují měnit stavy. Místa, příslušící danému přechodu jsou spojeny hranami (arcs). Hlavní výhoda Petriho sítí, jako modelovacího nástroje je možnost grafického vyjádření a možnosti simulovat graficky dynamické chování navrženého modelu.

Petriho sítě v roce 1962 poprvé představil C.A. Petri ve své disertační práci. Postupem času se ukázalo, že se jedná o jeden z nejlepších a nejvhodnějších jazyků pro popis, modelování a analýzu systémů, ve kterých se vyskytují synchronizační, komunikační a zdroje sdílející procesy. V praktických příkladech použití Petriho sítí se zjistily dvě nevýhody.

První nevýhodou v Petriho sítích je chybějící koncept práce s daty, což má za následek, že vznikající modely se stávají nepřiměřeně velké, protože veškerá manipulace s daty musí být reprezentována přímo do struktury sítě.

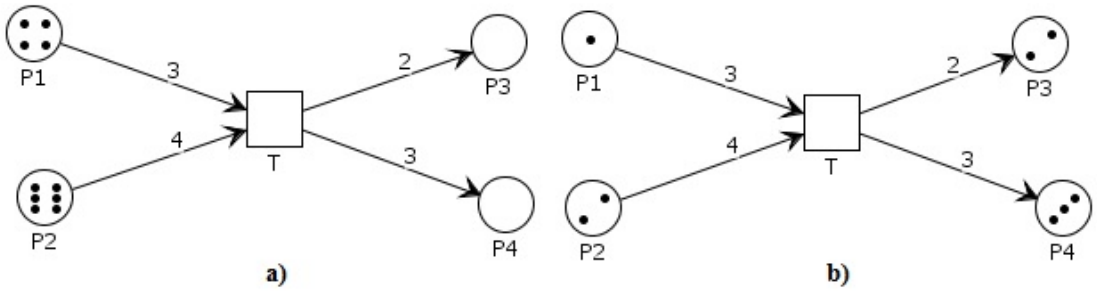
Druhou nevýhodou je chybějící hierarchický koncept a proto není možné stavět velké modely z množiny menších sub-modelů s dobře definovaným vzájemným rozhraním. Během rozvoje Petriho sítí se podařilo odstranit tyto nevýhody. Barevné Petriho sítě (Coloured Petri Nets) obsahují integraci datových struktur i možnost hierarchické dekompozice. [1]

Strukturu Petriho sítě detailněji popisují následující objekty [2] :

- Místa (places) které jsou graficky reprezentovány kružnicemi, místa mohou mít uvedenou kapacitu (maximální počet značek (tokens), které může dané místo obsahovat) pokud místo nemá uvedenou kapacitu, považuje se jeho kapacita za neomezenou.
- Přechody (transitions) graficky reprezentovány obdélníky (někdy čtverci).
- Orientovanými hranami (arcs), graficky reprezentovány šipkami, které směřují od místa k přechodu, nebo z přechodu do místa. Hrany mohou mít uvedenu váhu, která udává násobnost hrany. Pokud hrana nemá uvedenou váhu, její váha je rovna jedné.

V případě *ohodnocení hran* [1] , je každá hrana ohodnocena přirozeným číslem, jedná se o minimální počet značek, které musí vstupní místo obsahovat, jedná se tedy o vstupní podmínku pro událost. Odpalem přechodu odebereme ze vstupního místa tolik značek, kolik je hodnota hrany spojující místo s přechodem a do každého výstupního místa

přidáme tolik značek, kolik je hodnota příslušné hrany. Tohle zobecnění se využívá při modelování požadavků na omezení zdroje systému.



Obrázek 1: Odpálení přechodu v Petriho síti s ohodnocenými hranami. Na obrázku a) vidíme stav před odpálením přechodu T a na obrázku b) stav po odpálení přechodu T.

Modelování [2] je využíváno pro získávání nových poznatků a informací o chování modelovaného systému. Díky poznatkům získaných během modelování je možné zaměřit se na podstatné rysy systému a lépe pochopit studovaný problém. Jedná se tedy o proces výběru modelu, jeho sestavení, přetváření, hodnocení a následně přechod zpět od modelu k reálnému systému. Modelování je specifická forma experimentu, která umožňuje poznávat objektivní principy sledovaného systému. S modelováním úzce souvisí i simulace. *Simulace* [2] slouží k účelům lepšího pochopení sledovaného systému, nebo za účelem posouzení různých variant činnosti systému.

Model [2] je zjednodušená reprezentace určitého reálného systému. Model reprezentuje chování, fyzikální a chemické vlastnosti, geometrii, statické a dynamické vlastnosti pozorovaného systému.

2.1 Varianty Petriho sítí

V současné době existuje spousta variant Petriho sítí, které mají svou specifickou vyjadřovací sílu a modelovací schopnosti. Všechny tyto varianty rozšiřují základní matematický model Petriho sítě. Petriho sítě se postupně vyvíjeli, aby jejich modelovací schopnost umožnila co nejvíce vyhovět praktickým potřebám. Mezi další varianty Petriho sítí patří následující typy [2] :

- *C/E Petriho síť (Condition/Event Petri net)*,
- *Petriho sítě s prioritami (Petri Nets with Priorities)*,
- *Časované Petriho sítě (Timed Petri Nets)*,
- *Barevné Petriho sítě (Coloured Petri Nets)*,
- *Hierarchické Patriho sítě (Hierarchical Petri Nets)*,

- *Objektové Petriho sítě (Object-Oriented Petri Nets)*,
- *P/T Petriho síť (Place/Transitions Petri Net)*.

2.2 Výhody Petriho sítí

Petriho síť se především používá jako nástroj pro návrh, modelování a formální analýzu dynamických systémů.
Mezi hlavní výhody použití Petriho sítí pátří [2] :

- Grafická reprezentace Petriho sítí je srozumitelná i člověku, který není obeznámen s detaily teorie Petriho sítí.
- Petriho sítě jsou dostatečné obecné, tudíž mohou být použity pro velké množství různých systémů.
- Petriho sítě umožňují oproti ostatním jazykům pro popis systémů, jak popis stavů, tak i akcí.
- Možnost konstruovat rozsáhlé modely Petriho sítí pomocí menších modelů Petriho sítí.
- Graf, který popisuje Petriho síť dovoluje snadné vyjádření jevů, jako např. synchronizace, vzájemné vyloučení a paralelní výpočty.
- Možnost implementovat simulátory chování Petriho sítí.

3 MPI

Průmyslový standart pro programování výpočetních aplikací v oblasti systémů s distribuovanou pamětí je Message Passing Interface[1]. Jedná se o knihovnu, která implementuje message-passing protokol pro paralelní programování. Tato knihovna se zaměřuje především na paralelní programování, kde se data z adresového prostoru jednoho procesu posílají na druhý proces prostřednictvím vzájemné spolupráce mezi sebou. MPI umožňuje provádět kolektivní operace mezi procesy a tyto operace jsou v MPI reprezentovány jako funkce.

Programy vytvářené tímto způsobem mohou být spuštěny na multiprocesorech s distribuovatelnou pamětí, sítích pracovních stanic a kombinaci těchto možností. Dále je možné použít implementace se sdílenou pamětí, včetně těch pro vícejádrové procesory a hybridní architektury. MPI má tyto charakteristické vlastnosti [4] :

- Každý proces má své vlastní proměnné.
- Procesory si mezi sebou navzájem posílají zprávy s daty.
- Každý procesor má své jedinečné pořadové číslo.
- Program je obvykle napsán v programovacím jazyce C nebo Fortran.
- Univerzální, nezávislé na architektuře.

Norma MPI nespecifikuje operace, které vyžadují větší podporu operačního systému, než je běžný standart. Do těchto operací patří například, vzdálené spuštění, nebo aktivní zprávy. Dále standart nezahrňuje nástroje pro tvorbu programů a jejich ladění. [4]
V MPI rozlišujeme dva základní druhy komunikace. Jedná se o point-to-point komunikaci a globální(kolektivní) komunikaci. [3] [4] Tyto komunikace jsou popsány v následujících kapitolách.

3.1 Point-to-point komunikace

Jedná se o nejvíce používaný komunikační vzor v MPI. Point-to-point komunikace zahrňuje všechny metody MPI, které slouží pro přenos zprávy mezi dvojicí procesů. MPI nabízí spoustu funkcí, pro point-to-point komunikaci, které se od sebe nepatrně liší a mohou mít vliv na výkon MPI programu.

Point-to-point komunikace probíhá ve formě, kdy proces, který odesílá zprávu s daty odešle MPI_SEND a následně proces, který přijmul data odešle odpověď ve formě MPI_RECV. Oba procesy musí vědět se kterým procesem komunikují, tj. zdroj nebo cíl zprávy. Zpráva obsahuje identifikační tag a typ dat, které jsou předány.

Typicky se tento druh komunikace používá na architekturách typu master-slave. Kdy master je tzv. řídící uzel, který zodpovídá za řízení svých "sluhů".

[1] http://www.mpi-forum.org/

V oblasti point-to-point komunikace MPI dále poskytuje následující dva druhy komunikace:

- *Blokující komunikaci* - Blokující komunikace znamená, že proces čeká do doby, než obdrží zprávu s daty, poté může pokračovat ve zpracování dat.
- *Neblokující komunikaci* - Komunikace, která nezpůsobuje pozastavení programu. Program nesmí modifikovat odesílaná data, dokud komunikace neskončí. Během komunikace program může stále vykonávat kód.

3.2 Kolektivní komunikace

Kolektivní, někdy též nazývaná jako globální komunikace se liší v mnoha ohledech od point-to-point komunikace. Jedná se o koordinovanou komunikaci v rámci skupiny procesů, která probíhá prostřednictvím komunikátoru MPI. Při použití této komunikace je nutné specifikovat hlavní(root) proces, který vytváří, nebo přijímá data v určité oblasti. V případech, ve kterých je použita složitá point-to-point komunikace, dochází k nahrazení za kolektivní komunikaci, navíc kolektivní komunikace nemusí být označena tagy. Typová shoda pro kolektivní operace je více striktní, než je tomu u point-to-point komunikace. Zejména pro hromadné operace platí, že množství odeslaných dat, musí přesně odpovídat množství specifikované přijímačem.

Mezi základní kolektivní operace patří následující operace [4] :

- **MPI_BARRIER:** Bariéra synchronizuje všechny členy ve skupině.
- **MPI_BCAST:** Vysílání z jednoho člena všem členům skupiny.
- **MPI_GATHER:** Sběr dat od všech členů skupiny k jednomu členu.
- **MPI_SCATTER:** Rozloží data z jednoho člena na všechny členy skupiny.
- **MPI_ALLGATHER:** Variace Gatheru, kde všichni členové skupiny obdrží výsledek.
- **MPI_ALLTOALL:** Rozhodí, nebo shromáždí data ze všech členů všem členům skupiny. Tato operace se též nazývá kompletní výměna.
- **MPI_ALLREDUCE:** Globální reduktivní operace, jako je suma, maximum, minimum, nebo uživatelem definovaná funkce, kde výsledek je vrácen všem členům ve skupině, nebo pouze jednomu členu skupiny.
- **MPI_REDUCE_SCATTER_BLOCK:** Jedná se o kombinaci redukční a scatter operace.

4 Kaira

Kaira je open-source vývojové prostředí pro vytváření MPI aplikací. Hlavní myšlenkou pro vývoj bylo použít vizuální model inspirovaný barevnými Petriho sítěmi. Tento model je navržen tak, aby zachytil paralelní aspekty a komunikaci uvnitř vyvíjené aplikace.

Aplikace nemusí být programovány pouze vizuální formou, ale mohou být obohaceny i o jakýkoliv sekvenční C++ kód. Kaira umožňuje uživateli vytvářet MPI aplikace, spuštět simulace, nebo spustit state-space analýzu. Hlavním cílem je zjednodušit a zlepšit přístup paralelního programování distribuovaných paměťových aplikací v oblasti vědy a inženýrských výpočtů. Hlavní vlastnosti nástroje kaira jsou následující. [6]

- *Prototypování* - Uživatel je schopný vytvořit pracovní verzi vyvíjené aplikace v takové formě, která mu umožňuje experimentovat, jednoduše ji upravovat a vyhodnocovat její výkon.
- *Sjednocení* - Všechny činnosti v průběhu vývoje jsou kontrolovány a prezentovány ve stejném okně.
- *Aplikovatelnost* - Hlavním výstupem nástroje Kaira je skutečná aplikace, která může být spuštěná na paralelním počítači.
- *Integrace* - Existující sekvenční kódy se dají opakovaně pužít ve vývíjené aplikaci. Nástroj je také schopen vytvářet knihovny, které lze volat v jakékoliv sekvenční aplikaci.

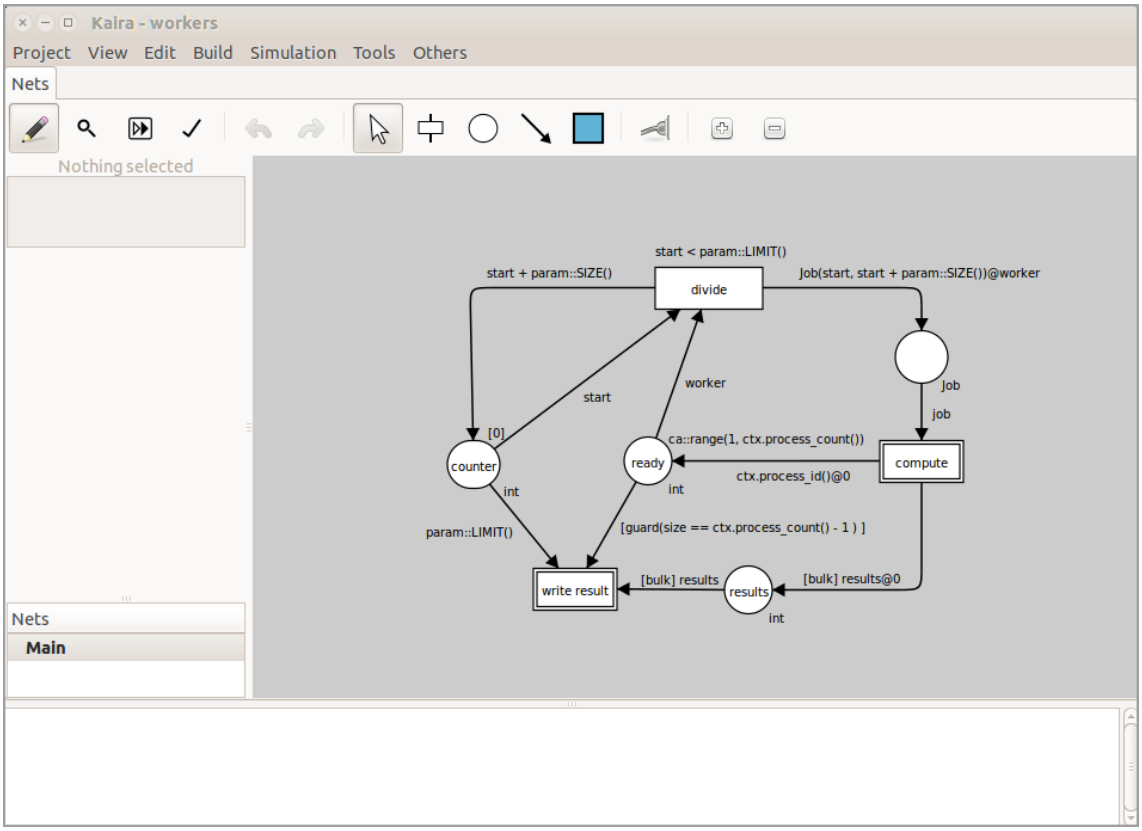
Pro zkušeného progrmátora MPI, může Kaira sloužit jako prototypovací nástroj s lehce dostupnými funkcemi, jako je profilování a analýza výkonu. Pro běžného uživatele v oblasti paralelního programování, který chce občas využít superpočítač bez velké časové investice, představuje Kaira vývojové prostředí s kontrolou nad běžnými aktivitamy, které vznikají v průběhu vývoje. [7]

Kaira se skládá ze tří hlavních částí [5] :

- *GUI* - Uživatelské rozhraní, které umožňuje pracovat se sítěmi a zdrojovými kódy, řídit simulace a zobrazovat výsledky analýz. Ukázka uživatelského rozrhaní je zobrazena na obrázku 2.
- *PTP* - Jedná se o hlavní část Kairy. Je to kompilátor, který překládá sítě do C++ kódu.
- *Knihovny* - Kaira obsahuje šest C++ knihoven, které jsou součástí výsledných programů. V první řadě se jedná o knihovnu *Cailie*, která je vždy součástí vygenerovaného programu. *CaVerif*, je součástí tehdy, pokud byla provedena state-space analýza. *CaSimrun* je spojena s výsledným programem, provedl-li uživatel predikci výkonu. *CaOctave* je součástí, pokud je použit *OCTAVE*. *CaClient* a *CaServer* jsou použity, pokud je sestavena RPC knihovna.

Základní vlastnost Kairy jako vývojového prostředí je vytvářet spustitelné aplikace. Výsledné aplikace mohou být vygenerovány ve třech režimech: MPI, vláknech, a sekvenčním režimu. MPI režim je považován jako hlavní. Jak název napovídá, tento režim produkuje aplikace, které používají MPI komunikaci. Ostatní dva režimy jsou určeny k využití podpůrných nástrojů, které nepracují dobře, nebo vůbec v distrubuovaném prostředí MPI. Oba režimy emulují chování MPI. Režim pracující ve vláknech emuluje MPI vrstvu pomocí PTHREADS místo samostatných procesů. V sekvenčním módu jsou aplikace prováděny postupně, takže i nástroje, které jsou určeny pro sekvenční aplikace mohou být použity. [5]

Tato vlastnost umožňuje snadné použití nástrojů, jako je GDB, nebo Valgrind k ladění sekvenčních částí aplikace. Kaira je zaměřena na ladění a analýzu paralelismu a komunikace, a předpokládá se, že tyto tyto nástroje se používají pro analýzu sekvenčního kódu v přechodech. Jakákoliv aplikace vytvořená v Kaiře, může být vygenerována ve všech třech režimech bez nutnosti změny v síti. Proces generování je plně automatický. [5]



Obrázek 2: Grafické uživatelské rozhraní nástroje Kaira.

5 Specifikace problému

Vývoj nové aplikace v nástroji Kaira, se provádí především ve vizuální podobě. Ve vizuální formě je uživatel schopný pomocí míst, hran a přechodů, snadno navrhnout požadované řešení. Ovšem s vizuálním návrhem souvisí problém se kterým se vývojář může setkat během vývojé své aplikace, neboť v praxi dochází k situacím, kdy uživatel během svého vývoje nové aplikace opakovaně vytváří již kdysi navržené sekvence míst, hran a přechodů, které splňují požadovanou funkcionalitu a jsou součástí jiných navržených aplikací. Tato skutečnost brzdí vývojáře v samotném vývoji nových aplikací v nástroji Kaira a snižuje tak jeho efektivitu.

V současné verzi nástroje Kaira nebylo možné tomuto problému vzdorovat. K řešení této specifikované situace, se kterou se uživatel setkává během vývoje nové aplikace, slouží knihovna, která obsahuje tyto opakující se sekvence míst, hran a přechodů, reprezentované ve formě pojmenovaných modulů. V knihovně jsou uloženy vybrané navrhnuté moduly, které reprezentují problematiku z konkrétní oblasti paralelního programování, resp. vizuálního návrhu vyvíjené aplikace.

Knihovna nemá pevně danou velikost, je možné knihovnu snadno rozšiřovat dle požadavků uživatele, což mu umožňuje přizpůsobit si nástroj Kaira tak, aby co nejvíce odpovídal jeho požadavkům při vývoji aplikací. Navrženým modulům je možné definovat vstupní místa, pomocí kterých je následně možné v průvodci vložení modulu do sítě provést konfigurovaci připojení modulu na požadované místa v síti. Takto vkládané moduly podstatně urychlí vizuální návrh vyvíjené aplikace.

Aby výše navrhnuté řešení bylo efektivní, bylo nutné vyřešit otázku univerzálnosti modulu. Univerzálnost modulů chápejme jako vlastnost, kdy se definice datových typů u jednotlivých míst modulu provede až při vkládání modulu do sítě, v průvodci vložení modulu. Podobnost můžeme nalézt u generik v programovacám jazyce C#, nebo jiných programovacích jazycích, kde při použití generik specifikujeme datový typ až ve chvíli konstrukce objektů.

Tato vlastnost je velice důležitá, neboť v situacích,ve kterých by uživatel chtěl modul použít s rozdílným datovým typem, než se kterým byl modul navrhnut, musel by celý modul zkonstruovat znovu, což by nepochybně popíralo hlavní motivaci pro vznik této práce, jelikož hlavním cílem této práce je usnadnit uživateli konstrukci vizuálních modelů a minimalizovat výskyt situací, kdy uživatel opakovaně kreslí totožné části řešení, které tvoří opakující se sekvence míst, hran a přechodů. Dále by začalo docházelo v knihovně k duplikacím modulů, akorát s rozdílem, že každý modul by pracoval s jinými datovými typy, ale vizuální část by zůstávala totožná, což je nežádoucí.

Univerzálnost nesouvisí pouze s datovými typy v místech modulu, ale také je možné této vlastnosti využít v kódu přechodu a headcode, neboť navržený modul nemusí být reprezentován pouze vizuální formou, ale může také obsahovat C++ kód, nezbytný pro správnou funkci modulu.

Veškerým uvedeným vlastnostem bylo nutné navrhnout vizuální podobu a implementovat je do grafického uživatelského rozhraní nástroje Kaira a to v takové formě, aby výsledné grafické rozhraní bylo co nejsrozumitelnější a snadné k použití. Pro mechanismus vložení modulu do uživatelem zvolené sítě jsem vytvořil průvodce, který

v několika krocích, provede připojení modulu na uživatelem požadované místa. Jednotlivé kroky, které je nutné provést při vložení modulu do uživatelem zvolé sítě jsou blíže specifikované v kapitole 8.

6 Modul

Modul chápejme jako obecnou šablonu, která po nastavení uživatelem v průvodci vložení modulu do sítě, poskytuje řešení konkrétní problematiky z oblasti paralelního programování.

O vytvoření modulu uvažujeme ve chvíli, kdy se uživatel setkává s opakující se síťí, která má konkrétní funkcionalitu, např. ve většině případech se uživatel setkává se situací, kdy potřebuje přiřadit data jednotlivým procesům. Právě v tomto případě se nabízí řešení v podobě vytvoření modulu, jehož hlavní funkce bude spočívat v přiřazování pracovních dat jednotlivým procesům.

Struktura a vlastnosti modulu jsou totožné se strukturou nového projektu v nástroji Kaira, kde hlavní část tvoří vizuální model a také může obsahovat c++ kód.

Je-li modul navržený jako šablona, potom není schopný překladu do chvíle, dokud neprojde nastavením v průvodci vložením modulu a následně vložením do pracovní sítě. Modul samozdřejmě může být vytvořen jako model schopný překladu, ovšem v tomto případě uživatel ztrácí univerzálnost modulu a modul je navržen pouze pro konkrétní datové typy.

Pro správnou funkci modulu jsou do Kairy přidány vlastnosti, které jsou nezbytné pro správné nastavení datových typů v modulu a k řízení mechanismu vkádání modulu do vizuálního modelu. Tyto mechanismy a postupy které je nezbytné dodržet při práci a návrhu modulů jsou dále podrobněji vysvětleny v následujících kapitolách.

Veškerá navigace je obsažena v menu nástroje Kaira, kde se nachází položka s názvem Project, která obsahuje hlavní funkce pro práci s moduly, jako vytvořit nový modul (New module), uložit modul (Save module) a načíst modul(Load module).

6.1 Vstupy modulu

Aby bylo možné modul z knihovny napojit na konkrétní místa v síti, musel být zaveden způsob jak označit místa modulu za vstupní. Klíčový znak pro tento způsob označení je znak $ tzv. dolar. Znak dolaru uvádíme v definici názvu místa. Za znakem dolaru může i nemusí následovat samotný název místa, neboť takto označené místo mechanismus připojení automaticky chápe jako vstupní místo modulu.

Místa, které jsou označená za vstupní budou následně nahrazeny místy z uživatelské pracovní sítě. Jinými slovy, jedná se o místa, které nám zajistí správné napojení modulu do uživatelem požadovaných míst v uživatelské pracovní síti. Samotná konfigurace připojení modulu se provádí v průvodci vložení modulu, který je blíže specifikován v kapitole 8.

Vstupní místa lze připojit pouze na místa v síti, nikoliv na přechody. Místa, na které lze připojit modul, nejsou ničím omezena a je pouze na uživateli, jak nakonfiguruje připojení modulu v průvodci.

6.2 Univerzálnost modulů

Modul by měl být navržen co nejobecněji, aby bylo možné s modulem pracovat v co nejvíce případech, aniž by bylo nutné měnit jeho strukturu. Z tohoto důvodu bylo nutné sestrojit mechanismus, který by uměl nastavovat datové typy na označených místech, vzhledem k případům použití modulu. Jestliže by byl modul vytvořený s konkrétními předdefinovanými datovými typy, znamenalo by to, že modul by byl použitelný pouze v případech, ve kterých by datové typy požadavků na modul odpovídaly jeho návrhu. Jinými slovy, modul by byl použitelný pouze v případech, ve kterých by byla shoda v použitých datových typech.

Pokud by modul obsahoval přednastavené datové typy, potom by docházelo k situacím kdy by uživatel byl nucen přenastavovat datové typy míst vzhledem k požadavkům na modul, což by bylo zbytečně pracné a nepohodlné. Jedno z možných řešení je navrhnout moduly bez datových typů a nechat uživatele, ať provede nastavení datových typů dle jeho potřeb.

Mé řešení univerzálnosti modulů je inspirováno v generických typech, které známe z programovacích jazyků, jako je např. C#. Použitím generik získáváme nástroj, jak specifikovat datový typ až ve chvíli konstrukci (volání) objektů. V samotné třídě se poté pracuje s generickým typem, který slouží jako zástupce pro budoucí datový typ. Můžeme si to představit tak, že se generický typ ve třídě změní např. na int ve chvíli, kdy vytvoříme její instanci. Jedná se tedy o možnost třídy nějakým způsobem parametrizovat.

```
// Deklarace genericke tridy
public class GenericList<T>
{
    void Add(T input) { }
}
class TestGenericList
{
    private class ExampleClass { }
    static void Main()
    {
        // Deklarace listu typem int
        GenericList<int> list1 = new GenericList<int>();
        // Deklarace listu typem ExampleClass.
        GenericList<ExampleClass> list2 = new GenericList<ExampleClass>();
    }
}
```

Výpis 1: Ukázka práce s generiky v jazyce C# kde T je nahrazeno datovým typem. [9]

Možností nahradit proměnou T, za datovým typ jsem využil ve svém řešení, ovšem místo T jsem zvolil znak #. Provede-li uživatel v definici datového typu místa, označení začínající znakem #, znamená to, že na tomto místě, musí uživatel v rozhraní průvodce vložení modulu doplnit datový typ, který bude následně nahrazen v označeném místě. Za znakem # v inicializaci datového typu místa může následovat uživatelský tag, vztahující se ke konkrétnímu místu, výsledné označení může vypadat např. #myType. Je to z toho důvodu, aby mohl uživatel od sebe odlišit různé datové typy, protože pokud by

platilo pravidlo, že za označení # se doplní jeden zvolený datový typ, nebylo by možné ošetřit situace, kdy modul pracuje s více datovými typy, nebo ošetřit situace kdy dochází k přetypování apod.

Uživatel během vkládání modulu do sítě je následně vyzván, aby provedl nastavení datových typů na místo jednotlivých tagů. V případě, mám-li v síťi tag s označením #myType a provedu nastavení na int, potom veškeré místa, které obsahují tento tag, budou přetypována právě na zvolený int.

Zvláštní případy nastávají u míst, které jsou označeny jako vstupní místa modulu. V těchto případech nemusí být místa nabídnuta uživateli pro nastavení datového typu, neboť mechanismus je navržen tak, aby v případě vstupních míst modulu, převzal datový typ míst, na které se modul připojí. Obsahuje-li tedy modul vstupní místo, které má označený datový typ začínající znakem #, znamená to, že vstupní místo modulu skopíruje datový typ místa, na které je modul připojený.

V případě vložení modulu do nové sítě, nebo existující sítě, která neobsahuje místa, na které je možné připojit modul, potom je uživatel automaticky vyzván, aby provedl inicializaci datových typů u všech míst, které mají odpovídající označení začínající znakem #.

6.3 Headcode a kód v přechodu

Univerzálnost modulu nesouvisí pouze s místy, ale je možné použít tagové označení i v headcode projektu. Je-li v modulu místo, které obsahuje tagové označení v definici datového typu, je možné použít toto tagové označení například i jako datový typ atributu třídy definované v headcode modulu, neboť po inicializaci datových typů uživatelem v průvodci vložení modulu do sítě, dojde k nahrazení těchto tagů i v headcode. Obdobně je tomu tak i u přechodů, ve kterých uživatel chce doplnit kód. Doplnění kódu v přechodu souvisí s místy, které jsou přímo připojené k danému přechodu.

Obsahuje-li místo tagové označení, promítne se toto označení i v kódu přechodu. V přechodu je dále možné použít i tagové označení místa, které není přímo napojeno na daný přechod, neboť při inicializaci datových typů v průvodci vložení modulu, se inicialiace promítne nejen u míst a v headcode, ale i u přechodů.

Je-li součástí modulu kód, který je uložený v headcode, dojde po vložení modulu do sítě k přidání tohoto kódu do headcode uživatelova projektu.

7 Knihovna modulů

Při návrhu knihovny byl kladen důraz na zastoupení modulů, které reprezentují nejčastější situace se kterými si vývojář setkává během vizuálního návrhu aplikace. Knihovna obsahuje moduly, které zastupují oblast point to point komunikace a oblast kolektivní komunikace. Navržené moduly jsou zaměřeny na oblast přidelování práce jednotlivým procesům.

Hlavní vlastnosti knihovny:

- obsahuje uložené moduly,
- oddělení modulů od uživatelských projektů,
- rozšiřitelnost.

Primární role knihovny je shromáždit veškeré uložené moduly do jednoho místa, ze kterého bude možné realizovat načítání modulů do uživatelských vizuálních modelů. Pro tuto knihovnu jsem zvolil umístění v adresáři nástroje Kaira, která se nachází na adrese **Kaira/Modules**.

Zvolí-li uživatel v hlavním menu položku *Module* a následně *New module*, jeho modul bude automaticky přesměrován a uložen do této knihovny.

Další vlastnost knihovny spočívá v oddělení uživatelských projektů od modulů. Jelikož má modul totožnou strukturu jako běžný projekt, bylo nutné separovat moduly od projektů, aby došlo k jasnému rozezání co je modul a co není. Díky tomu má užovatel přehled o všech jeho dostupných modulech na jednom místě.

Vzhledem k tomu, že si uživatel může vytvářet moduly sám, není limitován pouze standartním obsahem knihovny.

Chce-li uživatel rozšířit knihovnu o další modul, může tak učinit dvěma způsoby. Standartní způsob vytvoření nového modulu je pomocí hlavní nabídky v nástroji Kaira a nebo existuje druhá možnost, skopírovat si modul od jiného uživatele, ovšem v tomto případě se musí uživatel ujistit, že opravdu kopíruje modul do správného adresáře. Je-li modul uložen na jiné adrese, než na adrese knihovny, nebude modul načten v nabídce pro výběr modulu v průvodci vložení modulu a nebude tedy možné modul vložit do uživatatelské sítě.

7.1 Modul workers

Modul workers zastupuje oblast point to point komunikace. Návrh modulu je inspirován ve stejnojmeném projektu, který je součástí nástroje Kaira v adresáři *Samples*. Následující text popisuje funkci a strukturu modulu.

Vstupní místo obsahuje list tokenů(uživatelská data), které jsou fyzicky uloženy jako `TokenList<T>`, kde T je stejného datového typu jako místo, na které je modul připojen, nebo jej definuje uživatel v průvodci vložení modulu, záleží na dané situaci, viz kapitola 6.2.

Počet tokenů, které budou přiřazeny jednotlivým procesům uživatel definuje v místě s názvem *Size*.

Místo s názvem *Empty* slouží k ošetření proveditelnosti přechodu *Assign* za běhu programu. Toto místo udržuje přechod *Assign* aktivní do doby, dokud nedojde k rozdělení veškerých tokenů. Po rozdělení vstupních tokenů dojde k nastavení místa *Empty* na hodnotu jedna, což způsobí zablokování přechodu. Přechod *Assign* obsahuje c++ kód, který je uveden v následujícím výpise.

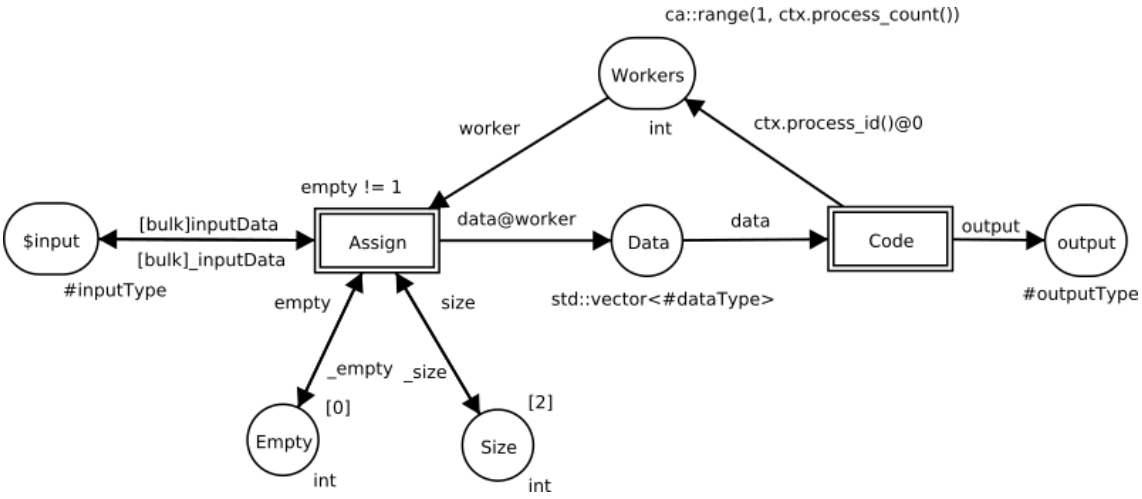
```
struct Vars {
    int &_empty;
    ca::TokenList<#inputType > &_inputData;
    int &_size;
    std :: vector<#dataType> &data;
    int &empty;
    ca::TokenList<#inputType > &inputData;
    int &size;
    int &worker;
};

void transition_fn (ca::Context &ctx, Vars &var){
    if (var.inputData.is_empty()){
        var._empty = 1;
        var._size = 0;
    }
    else{
        ca::Token<#inputType > *t;
        int counter = 0;

        for(t = var.inputData.begin(); t!=NULL; t = var.inputData.next(t)){
            if (counter < var.size){
                var.data.push_back(t->value);
            }
            else{
                var._inputData.add(t->value);
            }
            counter++;
        }
        if (var._inputData.is_empty()){
            var._empty = 1;
            var._size = 0;
        }
        else{
            var._empty = 0;
            var._size = var.size;
        }
    }
}
```

Výpis 2: Kód uvnitř přechodu Assign.

O reprezentaci jednotlivých procesů se stará místo *Workers*. Hlavním procesem byl zvolen proces 0, který plní funkci přidělování práce ostatním procesům, z těchto důvodů je místo *Workers* inicializováno od prvního procesu do posledního dostupného procesu. Počet dostupných procesů je v Kaiře možné zjistit pomocí funkce `ctx.process_count()`. Funkce vrací celé číslo (int) , které reprezentuje počet dostupných procesů.

Funkce modulu je rozdělena do dvou částí. V první části jsou přiděleny tokeny jednotlivým procesům a v druhé části jsou následně provedeny požadující operace. Na výstupu uživatel obdrží již zpracovaná data. Přechod *Assign* přiděluje konkrétní počet tokenů (počet je dán místem *Size*) jednotlivým dostupným procesům.

Přechod *Code* slouží k doplnění uživatelského c++ kódu. V tomto přechodu jsou soustředěny operace, které zpracovávají data na jednotlivých procesech.



Obrázek 3: Vizuální podoba modulu Workers.

7.2 Modul scatter

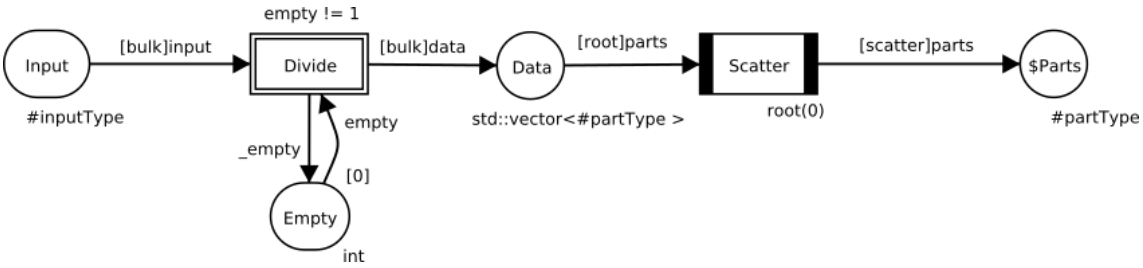
Modul je pojmenován Scatter, neboť jeho součástí je operace Scatter. Scatter je funkce z oblasti kolektivní komunikace. Modul plní funkci rozesílání dat jednotlivým procesům.

Vstupní místo obsahuje data, která budou dále rozeslána dostupným procesům. V přechodu *Divide* dochází k rovnoměrnému rozdělení vstupních dat na dílčí části. Velikost těchto částí vychází z výpočtu velikosti vstupního listu poděleno počtem dostupných procesů. Tyto části jsou následně uloženy v místě s názvem *Data*, které slouží jako zdroj dat pro samotnou funkci Scatter. Na proveditelnost přechodu *Divide* dohlíží místo s názvem *Empty*, které testuje vstupní list jestli není prázdný. Jakmile dojde k situaci, kdy list je prázdný, potom je hodnota místa *Empty* nastavena na hodnotu 1, což následně zablokuje proveditelnost přechodu *Divide*. Předpokládané nasazení modulu je v situacích, kdy list nebude prázdný, neboť místo *Empty* je v počátečním stavu nastaveno na hodnotu 0. Kontrola proveditelnosti přechodu *Divide* nastává až po prvním provedení přechodu. Kód obsažený v přechodu *Divide* je zobrazen v následujícím výpise.

```
struct Vars {
    int &_empty;
    ca::TokenList<std::vector< std::vector<int> > > &data;
    int &empty;
    ca::TokenList<int > &input;
};

void transition_fn (ca::Context &ctx, Vars &var){
    ca::Token<int> *t;
    std::vector< std::vector<int> > vd;
    int size;
    int process;

    if ( (var.input.size()) != 0 ){
        size = var.input.size();
        process = ctx.process_count();
        int count = size / process;
        int counter = 0;
        t = var.input.begin();

        for(int i = 0; i < process; i++){
                std::vector<int> v;
                for(t; t!=NULL;){
                    if ( counter < count){
                        v.push_back(t->value);
                        counter++;
                    }
                    else{
                        break;
                    }
                    t = var.input.next(t);
                }
                vd.push_back(v);
                counter = 0;
        }
        var.data.add(vd);
        var._empty = 1;
    }
    else{
        var._empty = 1;
    }
}
```

Výpis 3: Kód uvnitř přechodu Divide.

Přechod s názvem Scatter hraje v tomto modulu zásadní roli. Již na první pohled je zřejmá odlišnost přechodu od standartní značky běžných přechodů. Je to z důvodu, že přechod reprezentuje operaci kolektivní komunikace s názvem Scatter, která implementuje `MPI_Scatter`. Funkce scatter musí splnit následující podmínku. Na vstupním místě Scatteru musí být vector a zároveň velikost vektoru musí být totožná s počtem dostupných procesů. Při provádění přechodu obdrží data vectoru na pozici p a následně jej pošle procesu p.



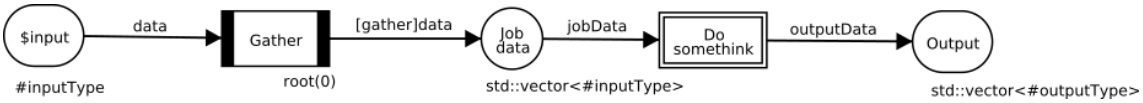
Obrázek 4: Vizuální podoba modulu Scatter.

7.3 Modul gather

Název modulu gather souvisí s operací Gather (`MPI_Gather`), která reprezentuje oblast kolektivní komunikace a je součástí tohoto modulu. Funkce Gather je inverzní funkcí k funkci Scatter. Cílem tohoto modulu je sesbírat data z dostupných procesů pomocí funkce Gather a následně v přechodu s názvem *Do somethink* provést požadované operace na získaných datech.

Vstupní místo obsahuje jednotlivé vektory, které slouží jako zdroj dat pro přechod Gather. Aby funkce gather fungoval správně, musí být splněny následující podmínka.

Vstupní místo *Input* musí mít typ `std::vector<T>`, kde T je typ výrazu. Proces provedení přechodu *Gather* probíhá tak, že root obdrží vstupní vektory, které sloučí do jednoho vektoru a tento vektor je poslán na výstup přechodu *Gather*, tedy na místo s názvem *Job data*. Na výsledném vektoru jsou poté provedeny operace, definové uživatelem v přechodu s názvem *Do somethink*. Výsledné data uživatel obdrží v místě s názvem *Output*.



Obrázek 5: Vizuální podoba modulu Gather.

8 Průvodce vložení modulu

Průvodce vložení modulu zapouzdřuje veškeré mechanismy, které jsou nutné pro práci s moduly. Průvodce se nachazí v nástrojové liště a je uživateli zobrazen po kliknutí na ikonku connect. Před tím, než bude modul vložen do uživatelské sítě, je nutné provést několik kroků.

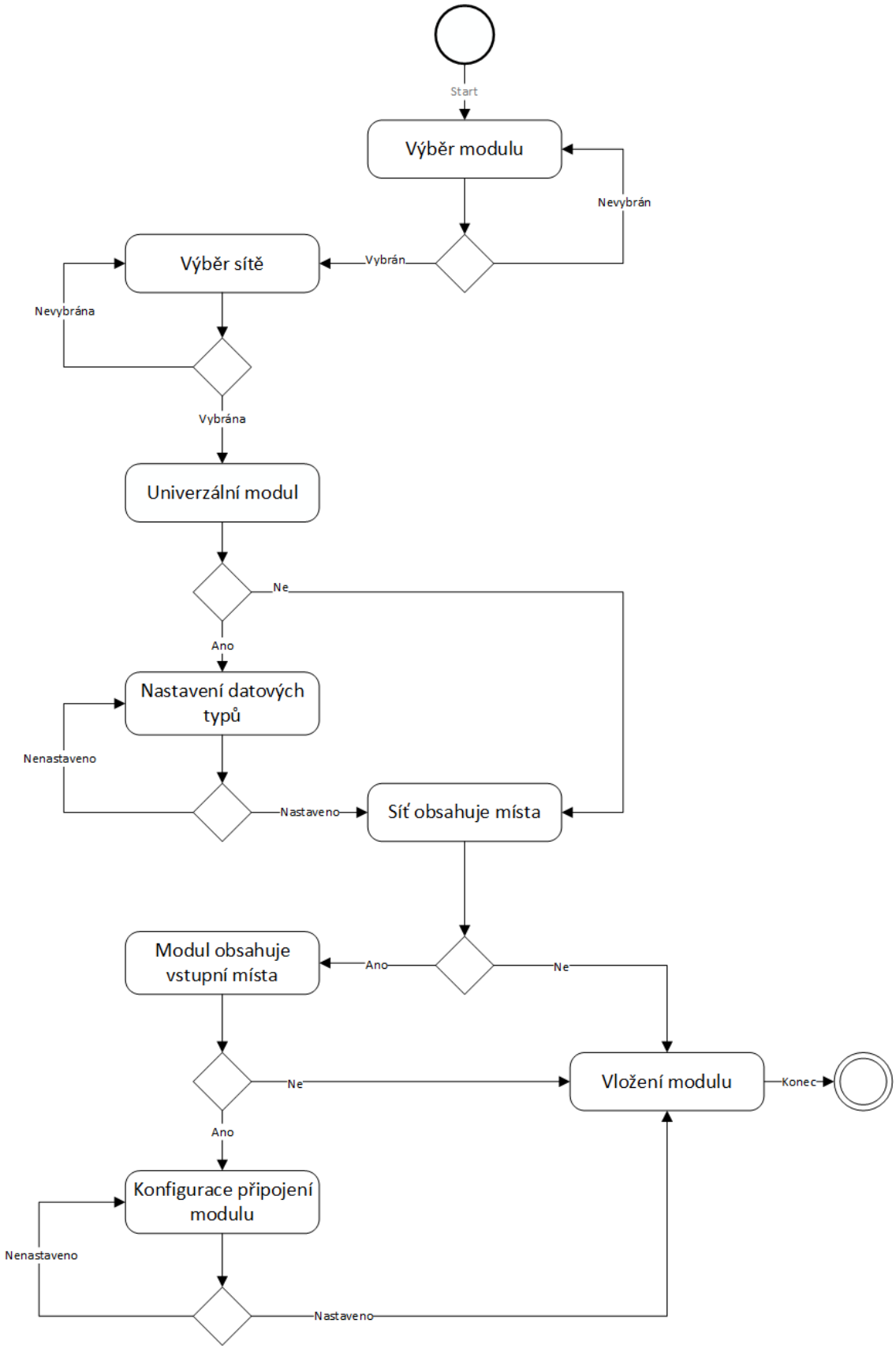
Proces vložení modulu je rozdělen na následující etapy:

1. Výběr modulu.
2. Výběr sítě.
3. Nastavení datových typů.
4. Konfigurace připojení modulu.

Proces vložení modulu do uživatelem zvolené sítě je zobrazen na následující straně, jako diagram aktivit. Zbylá část kapitoly se věnuje detailnějšímu popisu jednotlivých kroků. U některých kroků je uvedena i grafická podoba průvodce vložení modulu.

Při návrhu průvodce vložení modulu jsem se inspiroval ve standartních instalačních průvodcích se kterými se uživatel běžně setkává při instalaci softwaru. Ve spodní části hlavního okna se nachází ovládaccí panel s tlačítky, které umožňují uživateli se pohybovat v jednotlivých etapách vložení modulu.

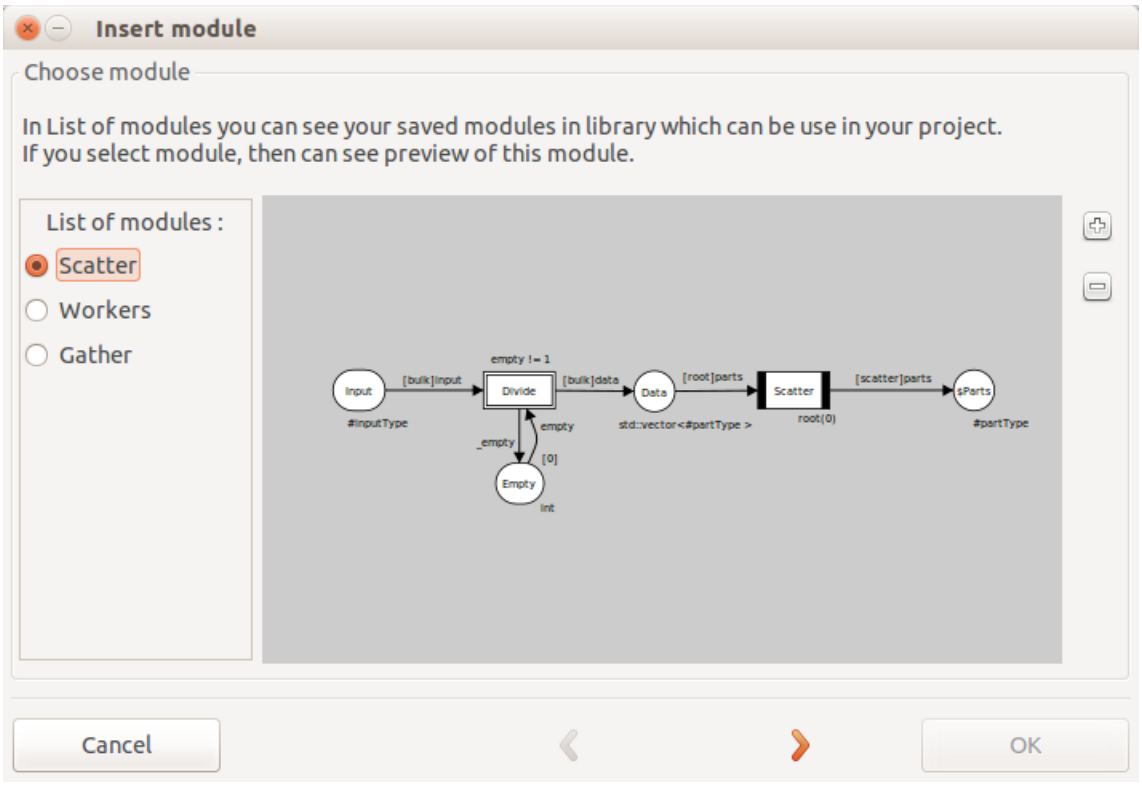
Mechanismus vložení modulu byl vyvíjen v programovacím jazyce Python. Pro tvorbu grafického rozhraní průvodce vložení modulu byla použita knihovna PyGTK [8] .



Obrázek 6: Proces vložení modulu zobrazený na diagramu aktivit.

8.1 Výběr modulu

V první etapě je uživatel vyzván, aby si zvolil modul z knihovny modulů, který chce vložit do uživatelské sítě.Jednotlivé dostupné moduly jsou zobrazeny v nabídce *List of modules*. Při kliknutí na požadovaný modul se uživateli zobrazí náhled modulu. Pokud není náhled čitelný, uživatel má možnost si modul oddálit, případně přiblížit.



Obrázek 7: Výběr modulu, v průvodci pro vložení modulu.

8.2 Výběr sítě

Ve druhé části vložení modulu je na uživateli, aby si zvolil síť do které chce modul vložit. Modul lze vložit do existující sítě aktuálního projektu, nebo použít modul k vytvoření nové sítě a ta je následně přidána do aktuálního projektu. V případě volby vložení modulu jako novou síť je následně uživateli zobrazeno dialogové okno, kde má uživatel možnost definovat název nově vytvořené sítě. Po definici názvu sítě je síť vložena do projektu a průvodce vložení modulu je ukončen.

8.3 Nastavení datových typů

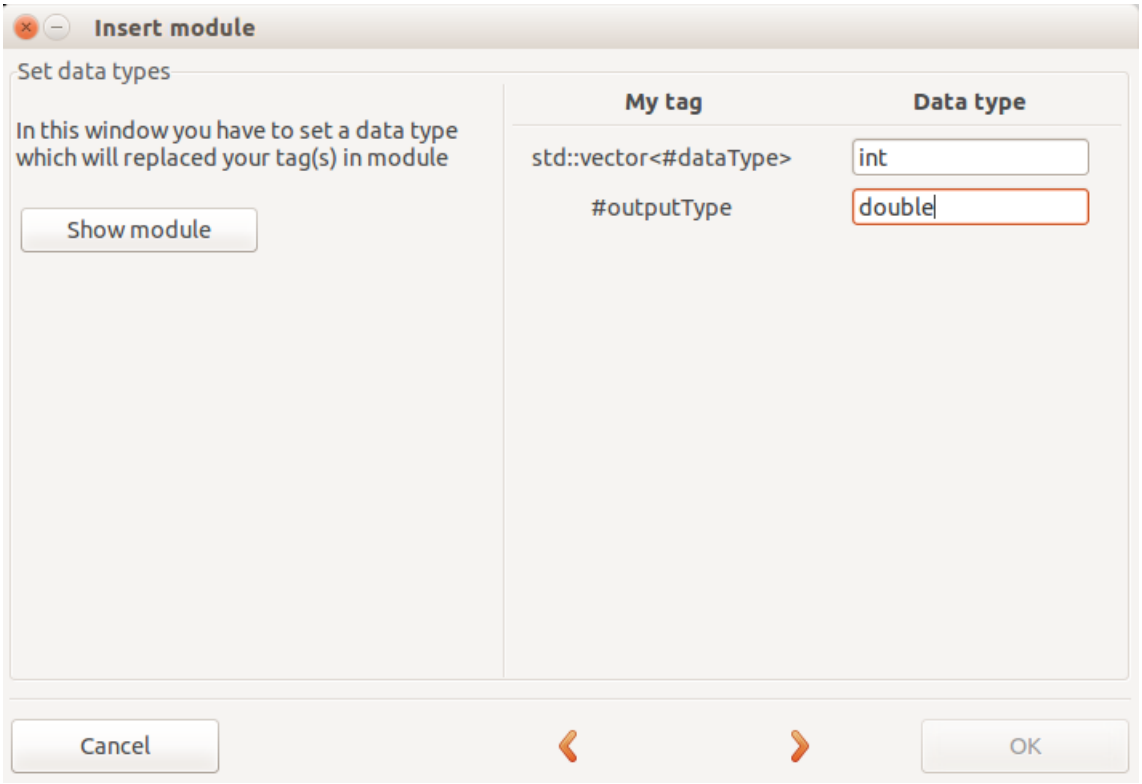
Po vybrání modulu a volby sítě, do které má být modul vložen, se uživatel dostává k nastavení datových typů. Pokud místo(a) modulu obsahují označení začínající tagem #, potom je uživatel před vložením modulu do sítě povinnen přiřadit konkrétním tagům datové typy, které jsou na základě uživatelské definice v průvodci následně nastaveny u míst v modulu.

Pravá část průvodce je věnována nastavení datových typů. Každý řádek obsahuje tagové označení míst v modulu a vedle něj kolonku pro přiřazení datového typu.

Speciální případy jsou vectory, kdy vector může mít uveden místo datového typu tagové označení a v tomto případě je tento tag pro přehlednost uveden spolu s vectorem.

Důležité je podotknout, že uživatel nastavuje datový typ za tagové označení, takže v případě, že se jedná o vector s označením `std::vector<#myType>`potom je zachován vektor a datový typ je pouze nahrazen za tagové označení. V případě přiřazení uživatelem k tagovému označení `#myType` například int, potom výsledný datový typ bude `std::vector<int>` . Nelze v tomto případě měnit celý datový typ, ten je respektován a vychází z návrhu modulu.

Pro lepší přehlednost a orientaci je v levé části tlačítko *Show module*, které zobrazí náhled modulu.



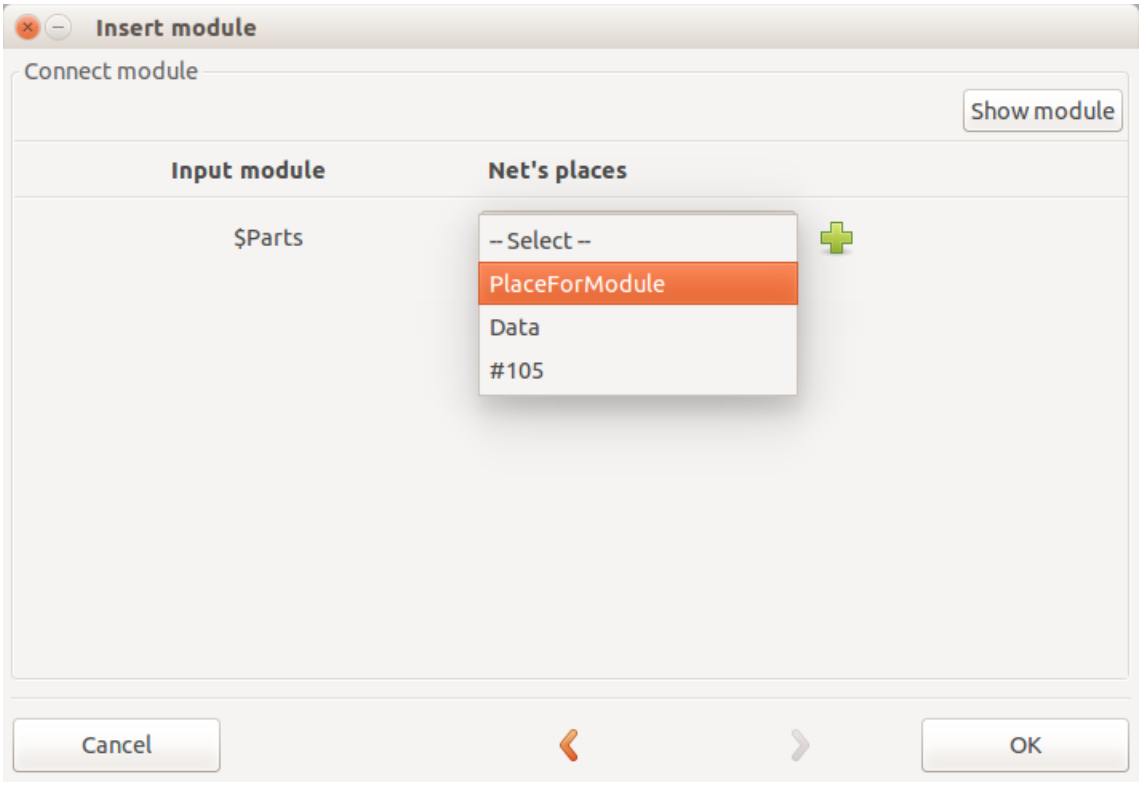
Obrázek 8: Nastavení datových typů v místech modulu.

8.4 Konfigurace připojení modulu

Poslední částí je samotná konfigurace připojení modulu. Mechanismus připojení modulu do konkrétních míst sítě je navržen tak, aby byl uživatel schopný jednoduše nakonfigurovat připojení modulu podle svých představ. V této sekci jsou uživateli zobrazeny vstupní místa modulu v levé části dialogového okna a ke každému vstupnímu místu modulu je standartně nabídnut jeden kombo box, který obsahuje místa (pokud nemají definované názvy, jsou reprezentovány jejich ID v síti) uživatelské sítě. Standartně je tedy nabídnuta možnost připojení modulu ve vztahu $1:1$ (místo v síti :vstupní místo modulu), ovšem pokud by uživatěl měl potřebu připojit více míst sítě na vstupní místo modulu, může tak provést tlačítkem $+$, které přidá uživateli další kombo box u konkrétního vstupu modulu, čímž je možné realizovat vztah až $1:N$, kde N je počet míst uživatelské sítě. Odebrání místa ve vztahu $1:N$ je možné provést ikonou X u jednotlivých kombo boxů.

Mechanismus konfigurace připojení modulu do sítě je dostupný pouze v případě, že síť do které má být modul vložen není prázdná (obsahuje místa na které je možné provést připojení). V případě, že síť je prázdná je modul pouze vložen do sítě.

Mechanismus kontroly připojení, v případě že síť není prázdná nedovolí uživateli kliknout na tlačítko *Ok* a provést tak vložení modulu, dokud uživatel neprovede konfiguraci připojení.



Obrázek 9: Konfigurace připojení modulu do sítě.

9 Příklad použití

Obsah této kapitoly je zaměřen na využití vytvořeného mechanismu a knihovny s moduly, na příkladu třídění dat.

Příklad na třídění dat jsem zvolil záměrně, neboť se jedná o typickou situaci, kdy uživatel disponuje vstupnímu daty uloženými v listu, kde v tomto případě list obsahuje řadu náhodných čísel a uživatel si přeje data setřídit. Aby při třídění byly využity všechny dostupné procesy, obdrží každý proces část dat, kterou setřídí a následně se provede setřídění všech dat na jednom procesu.

Řešení tohoto příkladu by se dalo rozdělit do několika částí.

V první části dojde k rozdělení vstupních dat na intervaly a následně jsou tyto intervaly přiřazeny dostupným procesům.

Ve druhé části jsou přiřazené intervaly na jednotlivých procesech setřízeny.

Třetí část provede sesbírání již setřízených intervalů z dostupných procesů a následně tyto intervaly sloučí do jednoho, který je následně setřízen. Takto setřízený interval je poté vytisknut do konzole jako výstup pro uživatele.

Pokud by uživatel postupoval standartním postupem, tedy nebylo by možné využít knihovnu modulů a mechanismus pro jejich vložení, musel by výše zmíněné části zkonstruovat.

V případě využití navrženého mechanismu společně s knihovnou modulů, dojde ke značnému zjednodušení celého postupu řešení.

Uživatel nemusí řešit rozdělení dat na intervaly a následně přiřazení na dostupné procesory a také nemusí řešit, jak sesbírá setřízené intervaly z procesorů a sloučí je do jednoho, aby tento interval následně setřídil. O tyto zmíněné postupy se postarají vložené moduly.

Postup řešení při použití modulů je následující.

Uživatel vytvoří přechod *Sort*, který bude provádět třídění jednotlivých intervalů, tento přechod musí být doplněn o c++ kód, obsahující buďto třídící algoritmus, nebo metodu *sort* z knihovny *std*. Kód v přechodu je uveden na následujícím výpise 4. Přechod obsahuje dvě místa, jedno vstupní(obsahuje intervaly nesetřízené) a druhé výstupní(obsahuje setřízené intervaly).

```
struct Vars {
    IntVector   &sor;
    std::vector<int> &uns;
};

void transition_fn (ca::Context &ctx, Vars &var)
{
    std::sort(var.uns.begin(), var.uns.end());
    var.sor = var.uns;
}
```

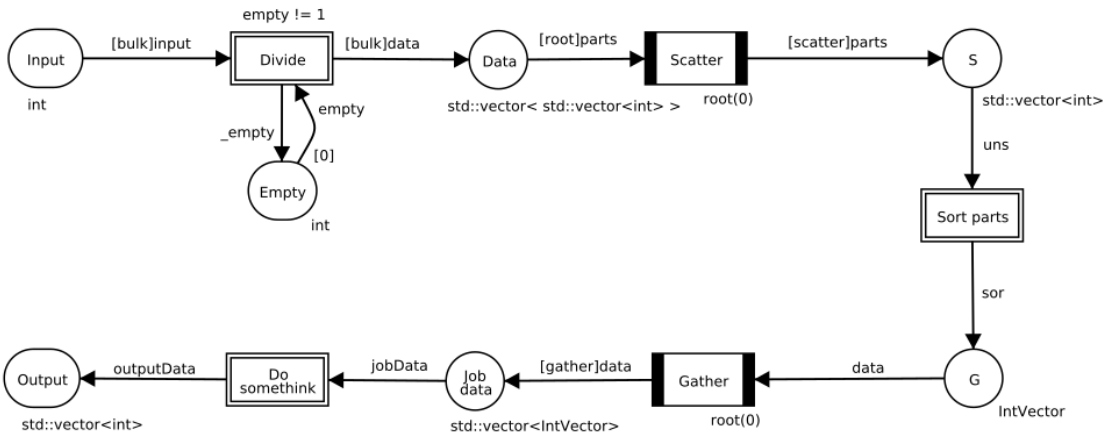
Výpis 4: Kód uvnitř přechodu Sort.

V tento moment uživatel vloží modul Scatter, který obstará rozdělení vstupních dat na itervaly o zadané velikosti a následně tyto intervaly přiřadí na dostupné procesy. Modul bude připojen na vstupní místo označené jako *S* na přechodu *Sort*. Následně uživatel vloží modul Gather, který provede sesbírání setříděných intervalů z dostupných procesů a tyto intervaly sloučí do jednoho, který následně setřídí v přechodu s názvem *Do somethink*. Kód uvnitř přechodu je uveden na výpise 5. Setřízená data budou poté vytisknuta do konzole. Tento modul bude připojen na výstupní místo označené *G* na přechodu *Sort*. Důležité je poznamenat, že typ *IntVector* na vstupním a výstupním místě přechodu Gather je ve skutečnosti `std::vector<int>` definový pomocí typedef, který se nachází v headcode příkladu a je zde z důvodu typové kontroly funkce Gather.

```
struct Vars {
    std::vector<IntVector> &jobData;
    std::vector<int> &outputData;
};

void transition_fn (ca::Context &ctx, Vars &var)
{
    std::vector< IntVector >::iterator row;
    IntVector:: iterator col;

    for(row = var.jobData.begin(); row!= var.jobData.end(); row++)
    {
            for(col = row->begin(); col!= row->end(); col++)
            {
                var.outputData.push_back(*col);
            }
    }

     std::sort(var.outputData.begin(), var.outputData.end());

     for (std::vector<int>::iterator it = var.outputData.begin(); it != var.outputData.end(); ++it)
     {
            printf ("%d ", *it);
     }
}
```

Výpis 5: Kód uvnitř přechodu Do somethink.

Výsledná síť po připojení modulů Gather a Scatter je zobrazena na následujícím obrázku.



Obrázek 10: Příklad použití modulu scatter a gather.

10 Možné rozšíření

Tato kapitola obsahuje náměty, které mohou být použity pro rozšíření navrženého mechanismu pro práci s moduly. Veškeré uvedené náměty vycházejí ze zkušeností, které jsem získal při řešení této práce.

10.1 Přehlednost

V současné situaci je modul vizuálně reprezentován celou svou strukturou (místa, přechody, hrany). Tato skutečnost je výhodná v tom, že uživatel na první pohled vidí celou strukturu modulu, což mu umožňuje přehledně pochopit funkcionalitu modulu a také případně snadno modifikovat modul. Naopak nevýhodu vidím v to, že jakmile dojde k vložení modulu do existující sítě, která již obsahuje uživatelskou síť, modul může zbytečně zabírat místo na plátně a také značně znepřehlednit celý projekt.

Možné rozšíření mé práce bych si dokázal představit v oblasti lepší přehlednosti sítě po vložení modulu. Vizi zpřehlednění popisuje následující text.

Vytvořit speciální značku, která jasně identifikuje, že se jedná o modul. Značka by mohla být ve tvaru přechodu, čili obdélník a například s označením M jako modul.

Pod touto značkou by byla uložena celá struktura modulu, kromě vstupních a výstupních míst. Vstupní místa musí být viditelné, neboť slouží k připojení modulu do sítě. Výstupní místa dále můžou sloužit jako zdroj dat pro další práci v síti.

Aby případně bylo možné provádět úpravy uvnitř modulu, bylo by vhodné vytvořit mechanismus, kdy kliknutím myši na značku modulu se objeví nabídka, která umožní užvateli zobrazit rychlý náhled modulu , nebo jeho editaci. V případě volby editace modulu by se v novém okně zobrazila celá struktura modulu a uživatel by byl následně schopen provádět úpravy modulu.

Otázkou zůstává, jak vyřešit proveditelnost takového modulu v módu simulací. Jedna z možností by byla navrhnout mechanismus, který by po kliknutí na modul automaticky provedl celý modul uvnitř navrhnuté značky.

10.2 Definice c++ kódu v přechodech modulu

Další možné rozšíření by se mohlo týkat průvodce vložení modulu do sítě. V mé bakalářské práci, průvodce umožní uživateli vložit modul do konkrétní sítě, na konkrétní místa a definovat datové typy, se kterými bude modul pracovat. Tohle je jakýsi základ, který průvodce dokáže.

Samotný návrh modulu je co nejobecnější, aby bylo možné použít modul v co největším počtu případů. Z tohoto důvodu, pokud si to situace žádá, modul obsahuje jen nejnutnější c++ kód, samotnou funkcionalitu, jakou uživatel potřebuje v přechodu, si musí doplnit sám.

Moje představa je taková, že uživateli by bylo umožněno doplnit požadovanou funkcionalitu u přechodů, které to umožňují(obsahují c++ kód) již v průvodci vložení modulu.

Otázkou ovšem zůstává, zda se nejedná pouze o zatěžování průvodce vložení modulu ve smyslu, že uživatel je nucen vyplňovat kód již v průvodci, přitom může kód doplnit přehledně až po vložení modulu do sítě.

Samotný design takového okna musí působit co nejpřehledněji, aby uživatel již v průvodci nebyl odrazen od definování c++ kódu.

Dokážu si představit okno, ve kterém budou zobrazeny jednotlivé přechody modulu a ve chvíli, kdy uživatel klikne na požadovaný přechod, bude zobrazen jednoduchý editor pro doplnění požadovaného kódu.

Jedna z možností je nechat uživatele aby se rozhodl, zda si přeje doplnit kód před samotným vložením modulu, anebo uživatel provede definici kódu až po vložení modulu do sítě.

Definice c++ kódu by musela být zařazena až na úplný závěr průvodce vložení modulu, neboť při definici kódu jsou důležitá místa připojená k přechodům, protože právě od míst a jejich datových typů se následně odvíjí možnost definice proměnných v daném přechodu.

10.3 Rozšíření knihovny

Samotná knihovna poskytuje prostor pro rozšíření. Určitě by bylo vhodné navrhnout další moduly z oblasti paralelního programování a poskytnout řešení na další oblast příkladů.

11 Jiné využití

Za předpokladu, že by bylo možné vytvořené moduly zaštítit pod navrženou značku (viz. kapitola 10.1), dalo by se toto řešení využít při návrhu rozsáhlých projektů. Při představě, že bych řešil opravdu rozsáhlý projekt, síť by se snadno stala nepřehlednou.

Z tohotu důvodů by bylo vhodné rozsáhlý projekt rozdělit na jednotlivé dílčí projekty. Každý projekt by obsahoval přidělenou úlohu, která by realizovala dílčí část rozsáhlého projektu. Jednotlivé projekty by uživatel uložil jako jednotlivé moduly a ve finální fázi by tyto moduly pouze napojil na sebe, což by představovalo finální řešení projektu.

Tímto postupem by došlo k značnému zpřehlednění celé sítě. Uživatelská síť by byla převážně tvořena moduly, které by reprezentovaly části řešení.

Díky vlastnosti zobrazit strukturu modulu, by uživatel mohl snadno zobrazit strukturu jednotlivých řešení, resp. strukturu modulu.

Ovšem, pokud bych se rozhodl takto řešit rozsáhlý projekt, narazil bych v módu simulací na problém. Tento mód umožňuje pozorovat chování navržené sítě. Uživatel v tomto módu má možnost krok po kroku simulovat běh programu.

Aby mohl uživatel pozorovat chování navržené sítě, musí mít zobrazenou strukturu sítě. Z tohoto důvodu není uvedené řešení úplně ideální, neboť tím, že strukturu modulu by reprezentovala pouze značka s označením M(modul), by uživatel neměl možnost si proklikat jeho strukturu.

V tomto případě by uživatěl na první pohled měl přehled pouze o modulech, nikoliv o jejich struktuře. Tento stav by byl žádoucí pouze v případě, že by uživatel nechtěl pozorovat chování uvnitř jednotlivých modulů. Pokud ano, možné řešení by mohlo být následujícího.

Pokud by tedy uživatel chtěl spustit simulace, muselo by dojít k tomu, že by všechna modula zpřístupnila svou strukturu, na které by následně bylo možné provést simulaci.

12 Závěr

Tato bakalářská práce je zaměřena na vizuální vývoj aplikací v nástroji Kaira. Cílem této bakalářské práce bylo vytvořit knihovnu obsahující pojmenované moduly, které reprezentují sekvence hran, míst a přechodů, se kterými se uživatel nejčastěji setkává při vývoji aplikací v nástroji Kaira. Knihovna obsahuje moduly u kterých jsem uznal za vhodné, že reprezentují nejčastější situace, se kterými se uživatel setkává při návrhu svých řešení. Zaměřil jsem se především na přidělování dat jednotlivým procesům. V knihovně jsou zastoupeny moduly z oblasti kolektivní komunikace, konkrétně moduly Gather a Scather. Díky vlastnosti inspirované v generických typech jsem dosáhl univerzálního použití modulů s různými datovými typy.

Bakalářská práce poskytuje nástroj, jak se během vizuálních návrhů v nástroji Kaira vyhnout opakované konstrukci sekvenci míst, hran a přechodů. Tím, že jsem vytvořil mechanismus práce s moduly, jsem poskytl vývojáři nástroj k tomu, aby si knihovnu v nástroji Kaira přizpůsobil vlastním potřebám, vytvořil si banku nejčastějších příkladů se kterými se setkává během vývoje svých projektů a tyto příklady následně vkládal do svých sítí. Pokud vývojář použije navrhnuté moduly, urychlí tím samotný vývoj vyvíjené aplikace.

Navrhnutý mechanismus jsem zpřístupnil uživateli ve formě průvodce vložení modulu do sítě, který se podobá běžným instalačním průvodcům, které známe z instalací nového software.

Veškerá moje práce probíhala v grafickém uživatelském rozhraní nástroje Kaira. Mechanismus pro práce s moduly jsem vytvořil v programovacím jazyce Python. Při programování grafického rozhraní jsem použil knihovnu PyGTK. Veškerý vývoj probíhal na platformě Linux - Ubuntu. Práce probíhala v samotné větvi repozitáře GitHub.

V úvodní teoretické části mé práce jsem se soustředil na představení nástroje Kaira. Kaira pracuje s knihovnou MPI, kterou jsem stručně charakterizoval v samostatné kapitole. V nástroji Kaira probíhá vývoj aplikací především vizuální formou, která je inspirována Petriho sítěmi. Petriho sítě jsou představeny z pohledu návrhu a modelování dynamických systémů.

Hlavní část mé práce je věnována knihovně, která obsahuje vybrané moduly. V kapitole věnované modulům jsem uvedl základni vlastnosti modulů, jejich univerzální použití a možnosti napojení modulu do uživatelem zvolené sítě na konkrétní místa v síti. Dále jsem detailněji popsal význam knihovny modulů a následně uvedl jednotlivé navržené moduly. U každého modulu jsem pro názornost uvedl i jeho vizuální podobu.

Navrhnutý mechanismus pro práci s moduly, jsem demonstroval na příkladu třídění dat. V příkladu jsem použil navrhnuté moduly Gather a Scatter.

Během řešení mé bakalářské práce jsem získal náměty, které lze použít pro rozšíření současného navrženého řešení a současně jsem uvedl myšlenku dalšího možného využití navrhnutého řešení při vývoji rozsáhlých projektů. Tyto náměty jsem specifikoval na závěr mé práce.

Pro mne samotného je přínos této bakalářské práce v získaných znalostech z oblasti tvorby a návrhu grafického uživatelského rozhraní v jazyce Python a použití knihovny PyGTK. Při použití verzovacího systému GitHub jsem získal představu o tom, jak probíhá vývoj software v týmu programátorů. Dále jsem si osvojil znalosti v oblasti paralelního

programování a to především díky nástroji Kaira a knihovně MPI. Při návrhů modulů jsem si vyzkoušel praktickou využitelnost Petriho sítí.

13 Reference

[1] VORÁČKOVÁ, Šárka, Martin PĚNIČKA a Jaroslav VESELÝ. *Úvod do modelování procesů Petriho sítěmi*. Praha: ČVUT, 2008. ISBN 978-80-01-03979-3.

[2] KOCHANÍČKOVÁ, Monika. *Petriho sítě* [online]. Olomouc, 2008 [cit. 2015-01-11]. Dostupné z: http://phoenix.inf.upol.cz/esf/ucebni/petriho_site.pdf.renamed

[3] Message Passing Interface (MPI). BARNEY, Blaise. LAWRENCE LIVERMORE NATIONAL LABORATORY. *High Performance Computing* [online]. 2014, last modified 15.12.2014 [cit. 2015-01-11]. Dostupné z: https://computing.llnl.gov/tutorials/mpi/

[4] *MPI: A Message-Passing Interface Standard Version 3.0: Message Passing Interface Forum* [online]. 2012 [cit. 2015-04-17]. Dostupné z: http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf

[5] BÖHM, Stanislav. *Unifying Framework For Development of Message-Passing Applications* [online]. Ostrava, 2013 [cit. 2015-03-29]. Dostupné z: http://verif.cs.vsb.cz/sb/thesis.pdf. Ph.D. Thesis. Faculty of Electrical Engineering and Computer Science, VŠB – Technical University of Ostrava.

[6] BÖHM, Stanislav, Marek BĚHÁLEK, Ondřej MECA a Martin ŠURKOVSKÝ. *Application and theory of Petri nets and concurrency: 35th International Conference, Petri nets 2014, Tunis, Tunisia, June 23-27, 2014. Proceedings*. 1st edition. 2014, pages cm. ISBN 33-190-7733-3. Dostupné z: http://dx.doi.org/10.1007/978-3-319-07734-5_22

[7] BÖHM, Stanislav, Marek BĚHÁLEK, Ondřej MECA a Martin ŠURKOVSKÝ. Visual programming of MPI applications: Debugging, performance analysis, and performance prediction. *Computer Science and Information Systems*. 2014, vol. 11, issue 4, s. 1315-1336. DOI: 10.2298/CSIS131204052B. Dostupné z: http://www.doiserbia.nb.rs/Article.aspx?ID=1820-02141400052B

[8] PyGTK: GTK+ for Python. *PyGTK: GTK+ for Python* [online]. [cit. 2015-03-29]. Dostupné z: http://www.pygtk.org/

[9] Generics (C# Programming Guide). *Generics (C# Programming Guide)* [online]. 2013 [cit. 2015-03-11]. Dostupné z: https://msdn.microsoft.com/en-us/library/512aeb7t.aspx

14 Seznam příloh

Součástí BP je přiložené CD, které obsahuje následující soubory:

1. Bakalářská práce ve formátu pdf.
2. Rozšířený nástroj Kaira o navrhnutý mechanismus, spolu s příkladem a knihovnou modulů.
3. Textový soubor readme, který obsahuje seznam souborů, ve kterých jsem pracoval.